# Introduction to Data Science

## Session 3: R and the tidyverse

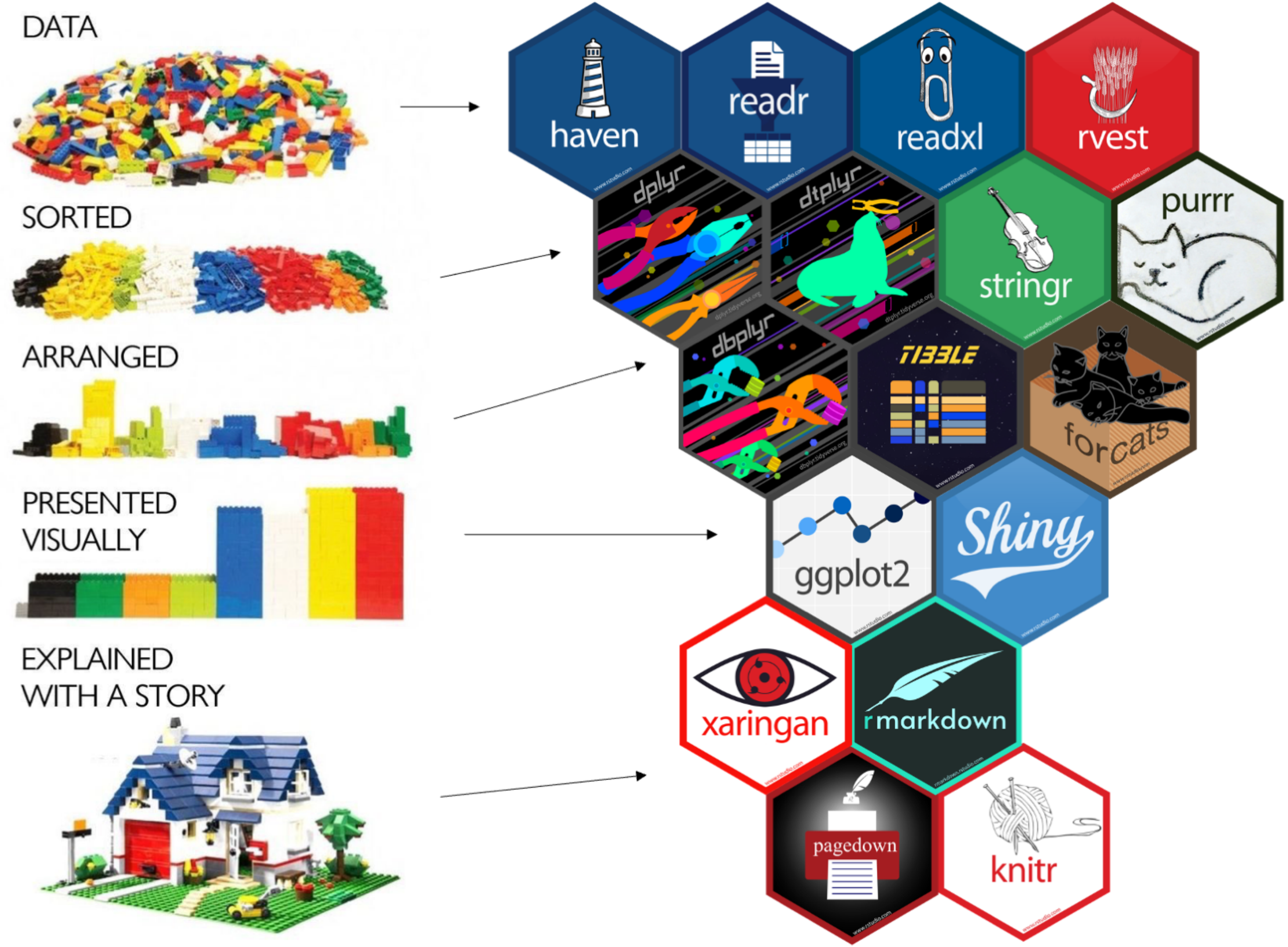
Simon Munzert
Hertie School | GRAD-C11/E1339

# Table of contents



[1] Parts of this lecture draw on materials from Grant McDermott's excellent *Data Science for Economists* class.

# Today's session in a nutshell



# Tidyverse basics

# What is the tidyverse?

## R packages for data science

- Let's take it from the tidyverse website:

**"The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures."**

- It's the contribution of many people of the R community.
- Hadley Wickham had a key role in shaping it by developing many of the core packages, such as `ggplot2`, `dplyr`, `tidyr`, `tibble`, and `stringr`.
- Install the complete tidyverse with:

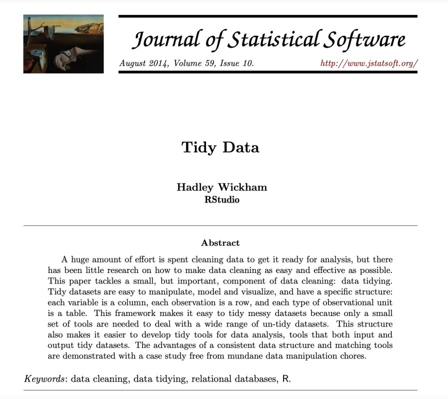
```
R> install.packages("tidyverse")
```

Hadley Wickham

# A guide to the tidyverse

## Valuable resources

- Welcome to the Tidyverse, a quick overview from many tidyverse contributors
- Tidy data, a foundational paper on data wrangling and structuring, by Hadley Wickham, 2014, *Journal of Statistical Software*; check here for a hands-on vignette based on the `tidyr` package
- The tidyverse design guide, a (soon-to-be book) manifesto to promote design consistency across the tidyverse
- R for Data Science, our main textbook for this course



*Journal of Statistical Software*

*August 2014, Volume 59, Issue 10.* *http://www.jstatsoft.org/*

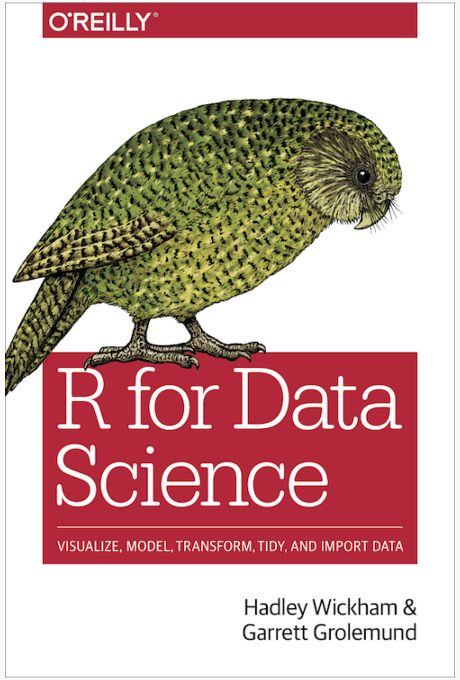
**Tidy Data**

**Hadley Wickham**
**RStudio**

**Abstract**

A huge amount of effort is spent cleaning data to get it ready for analysis, but there has been little research on how to make data cleaning as easy and effective as possible. This paper tackles a small, but important, component of data cleaning: data tidying. Tidy datasets are easy to manipulate, model and visualize, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table. This framework makes it easy to tidy messy datasets because only a small set of tools are needed to deal with a wide range of un-tidy datasets. This structure also makes it easier to develop tidy tools for data analysis, tools that both input and output tidy datasets. The advantages of a consistent data structure and matching tools are demonstrated with a case study free from mundane data manipulation chores.

*Keywords*: data cleaning, data tidying, relational databases, R.



# Tidyverse packages

## Loading the tidyverse

```
R> library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.3     ✓ dplyr   1.0.7
## ✓ tidyr   1.1.3     ✓ stringr 1.4.0
## ✓ readr   2.0.0     ✓ forcats 0.5.1
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

# Tidyverse packages

## Loading the tidyverse

```
R> library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.3     ✓ dplyr   1.0.7
## ✓ tidyr   1.1.3     ✓ stringr 1.4.0
## ✓ readr   2.0.0     ✓ forcats 0.5.1
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

- We see that we have actually loaded a number of packages (which could also be loaded individually): `ggplot2`, `tibble`, `dplyr`, etc.
- We can also see information about the package versions and some namespace conflicts.

# Tidyverse packages *cont.*

- In addition to the currently 8 core packages, the tidyverse includes many others for more specialized usage.[1]
- See here for an overview, or just in R directly:

```
R> tidyverse_packages()
```

```
##  [1] "broom"         "cli"           "crayon"        "dbplyr"
##  [5] "dplyr"         "dtplyr"        "forcats"       "googledrive"
##  [9] "googlesheets4" "ggplot2"       "haven"         "hms"
## [13] "httr"          "jsonlite"      "lubridate"     "magrittr"
## [17] "modelr"        "pillar"        "purrr"         "readr"
## [21] "readxl"        "reprex"        "rlang"         "rstudioapi"
## [25] "rvest"         "stringr"       "tibble"        "tidyr"
## [29] "xml2"          "tidyverse"
```

[1] It also includes a *lot* of dependencies upon installation. This is a matter of some controversy.

# Tidyverse packages *cont.*

- In addition to the currently 8 core packages, the tidyverse includes many others for more specialized usage.[1]
- See here for an overview, or just in R directly:

```
R> tidyverse_packages()
```

```
##  [1] "broom"         "cli"           "crayon"        "dbplyr"
##  [5] "dplyr"         "dtplyr"        "forcats"       "googledrive"
##  [9] "googlesheets4" "ggplot2"       "haven"         "hms"
## [13] "httr"          "jsonlite"      "lubridate"     "magrittr"
## [17] "modelr"        "pillar"        "purrr"         "readr"
## [21] "readxl"        "reprex"        "rlang"         "rstudioapi"
## [25] "rvest"         "stringr"       "tibble"        "tidyr"
## [29] "xml2"          "tidyverse"
```

- We'll use several of these additional packages during the remainder of this course (e.g., the `lubridate` package for working with dates and the `rvest` package for web scraping).
- However, bear in mind that these packages will have to be loaded separately.

[1] It also includes a *lot* of dependencies upon installation. This is a matter of some controversy.

# The tidyverse philosophy

## Key philosophy for tidy data

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Basically, tidy data is more likely to be long (i.e. narrow) format than wide format.

# The tidyverse philosophy

## Key philosophy for tidy data

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Basically, tidy data is more likely to be long (i.e. narrow) format than wide format.

## More unifying principles

- Today, the tidyverse stands for more than just "tidy data".
- It is guided by the principles of being **human centered**, **consistent**, **composable**, and **inclusive**.
- We will learn about these unifying principles inductively when working with more and more tidyverse packages.
- Later today, we will learn about tidyverse style principles of low-level code formatting.

# The tidyverse philosophy

## Key philosophy for tidy data

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Basically, tidy data is more likely to be long (i.e. narrow) format than wide format.

## More unifying principles

- Today, the tidyverse stands for more than just "tidy data".
- It is guided by the principles of being **human centered**, **consistent**, **composable**, and **inclusive**.
- We will learn about these unifying principles inductively when working with more and more tidyverse packages.
- Later today, we will learn about tidyverse style principles of low-level code formatting.

## Resources

Check out the tidyverse design guide for a comprehensive treatment of the tidyverse philosophy.

# Tidyverse vs. base R

# Tidyverse vs. base R: what's the difference?

- Both are compatible. You can wrangle your data with `dplyr`, plot it with `ggplot2`, and model it with yet another package.
- Ultimately, the tidyverse is just a bunch of (hugely popular!) packages that share design principles.
- Often, tidyverse packages don't reinvent the wheel. Instead, they offer more consistency in naming, arguments, and output (among other things).
- For instance, compare function naming principles (`tidyverse::snake_case` vs `base::period.case` rule; more on these conventions later) in these examples:

| tidyverse | base |
|---|---|
| `?readr::read_csv` | `?utils::read.csv` |
| `?dplyr::if_else` | `?base::ifelse` |
| `?tibble::tibble` | `?base::data.frame` |

- If you call up the above examples, you'll see that the tidyverse alternative typically offers some enhancements or other useful options (and sometimes restrictions) over its base counterpart.

# Tidyverse vs. base R: what's the difference?

- Both are compatible. You can wrangle your data with `dplyr`, plot it with `ggplot2`, and model it with yet another package.
- Ultimately, the tidyverse is just a bunch of (hugely popular!) packages that share design principles.
- Often, tidyverse packages don't reinvent the wheel. Instead, they offer more consistency in naming, arguments, and output (among other things).
- For instance, compare function naming principles (`tidyverse::snake_case` vs `base::period.case` rule; more on these conventions later) in these examples:

| tidyverse | base |
|---|---|
| `?readr::read_csv` | `?utils::read.csv` |
| `?dplyr::if_else` | `?base::ifelse` |
| `?tibble::tibble` | `?base::data.frame` |

- If you call up the above examples, you'll see that the tidyverse alternative typically offers some enhancements or other useful options (and sometimes restrictions) over its base counterpart.

- And **remember:** There are (almost) always multiple ways to achieve a single goal in R.

# Tidyverse vs. base R: what's the difference? *cont.*

**Tidyverse**

`Credit` sawiki.com

# Tidyverse vs. base R: what's the difference? *cont.*

**Tidyverse**

`Credit` sawiki.com

**Base R**


`Credit` multimedialab.be

# Tidyverse vs. base R: what to use?

## Stories from the past

- When I started to learn R ~13 years ago, there was no tidyverse. The learning curve felt much steeper. I often switched back to Stata for data wrangling.
- As the tidyverse grew, R became more convenient to use for the entire research pipeline.
- There's simply no need for you to live through the same pain.

## Why we start with the tidyverse

- Because clever people think it's the right way.
- Documentation + community support are great.
- Having a consistent syntax makes it easier to learn.

# Tidyverse vs. base R: what to use?

## Stories from the past

- When I started to learn R ~13 years ago, there was no tidyverse. The learning curve felt much steeper. I often switched back to Stata for data wrangling.
- As the tidyverse grew, R became more convenient to use for the entire research pipeline.
- There's simply no need for you to live through the same pain.

## Why we start with the tidyverse

- Because clever people think it's the right way.
- Documentation + community support are great.
- Having a consistent syntax makes it easier to learn.

## You still will want to check out base R alternatives later

- Base R is extremely flexible and powerful (and stable).
- There are some things that you'll have to venture outside of the tidyverse for.
- A combination of tidyverse and base R is often the best solution to a problem.
- Excellent base R data manipulation tutorials: here and here.

# Now, let's get started with the tidyverse!

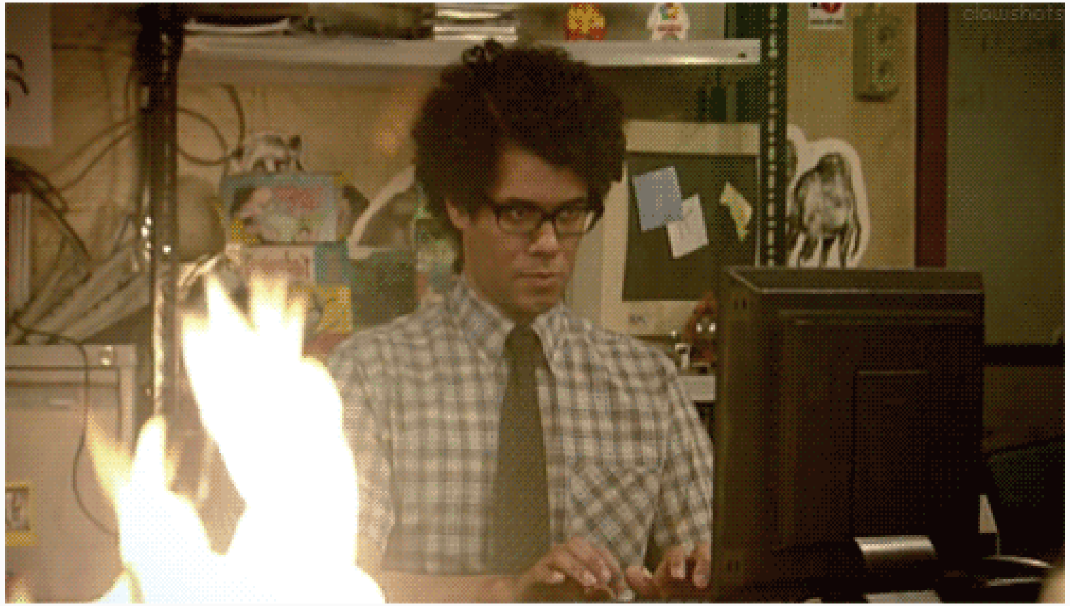
## R packages you'll need today

☑ **tidyverse**

☑ **nycflights13**

You can install/update them both with the following command.

```
R> install.packages(
+   c('tidyverse', 'nycflights13'),
+   repos = 'https://cran.rstudio.com',
+   dependencies  = TRUE
+ )
```



# Pipes


Credit likestowastetime/imgur

# The pipe

%>%

# Example

### The pipe way

```
R> Alex %>%
+   wake_up(7) %>%
+   shower(temp = 38) %>%
+   breakfast(c("coffee", "croissant")) %>%
+   walk(step_function()) %>%
+   bvg(
+     train = "U2",
+     destination = "Stadtmitte"
+   ) %>%
+   hertie(course = "Intro to DS")
```

### The classic way

```
R> hertie(
+   bvg(
+     walk(
+       breakfast(
+         shower(
+           wake_up(
+             Alex, 7
+           ),
+           temp = 38
+         ),
+         c("coffee", "croissant")
+       ),
+       step_function()
+     ),
+     train = "U2",
+     destination = "Stadtmitte"
+   ),
+   course = "Intro to DS"
+ )
```

# Example

### The pipe way

```
R> Alex %>%
+   wake_up(7) %>%
+   shower(temp = 38) %>%
+   breakfast(c("coffee", "croissant")) %>%
+   walk(step_function()) %>%
+   bvg(
+     train = "U2",
+     destination = "Stadtmitte"
+   ) %>%
+   hertie(course = "Intro to DS")
```

### The classic way, nightmare edition

```
R> alex_awake <- wake_up(Alex, 7)
R> alex_showered <- shower(alex_awake,
+                          temp = 38)
R> alex_replete <- breakfast(alex_showered,
+                            c("coffee", "croissant")
R> alex_underway <- walk(alex_replete,
+                        step_function())
R> alex_on_train <- bvg(alex_underway,
+                       train = "U2",
+                       destination = "Stadtmitte
R> alex_hertie <- hertie(alex_on_train,
+                        course = "Intro to DS")
```

# The beauty of pipes

## A simple but powerful tool

- The forward-pipe operator `%>%` pipes the left-hand side values forward into expressions on the right-hand side.
- We replace `f(x)` with `x %>% f()`.

# The beauty of pipes

## A simple but powerful tool

- The forward-pipe operator `%>%` pipes the left-hand side values forward into expressions on the right-hand side.
- We replace `f(x)` with `x %>% f()`.

## Why piping is cool

- It structures sequences of data operations as pipes, i.e. left-to-right (as opposed to from the inside and out).
- It serves the natural way of reading ("do this, then this, then this, ...").
- It avoids nested function calls.
- It improves cognitive performance of code writers and readers.
- It minimizes the need for local variables and function definitions.

# The beauty of pipes

## A simple but powerful tool

- The forward-pipe operator `%>%` pipes the left-hand side values forward into expressions on the right-hand side.
- We replace `f(x)` with `x %>% f()`.

## Why piping is cool

- It structures sequences of data operations as pipes, i.e. left-to-right (as opposed to from the inside and out).
- It serves the natural way of reading ("do this, then this, then this, ...").
- It avoids nested function calls.
- It improves cognitive performance of code writers and readers.
- It minimizes the need for local variables and function definitions.

## Background

- The pipe was originally created in 2014 by Stefan Milton Bache and published with the `magrittr` package.
- Magrittr? Get it? 🤡
- The basics come with the tidyverse by default, but `magrittr` can do more (watch out for the "tee" pipe, `%T>%`, the "exposition" pipe, `%$%`, and the "assignment" pipe, `%<>%`). Also, be sure to check out aliases.

# Piping etiquette

## When to avoid the pipe

- Pipes are not very handy when you need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- Don't use the pipe when there are meaningful intermediate objects that can be given informative names (and that are used later on).

# Piping etiquette

## When to avoid the pipe

- Pipes are not very handy when you need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
- Don't use the pipe when there are meaningful intermediate objects that can be given informative names (and that are used later on).

## Instead, here's how to use it

- `%>%` should always have a space before it, and should usually be followed by a new line.
- A one-step pipe can stay on one line, but unless you plan to expand it later on, you should consider rewriting it to a regular function call.
- `magrittr` allows you to omit `()` on functions that don't have arguments (as in `mydata %>% summary`). Avoid this feature.

# The base R pipe: |>

The magrittr pipe has proven so successful and popular that the R core team recently added a "native" pipe operator to base R (version 4.1), denoted `|>`.[1]

# The base R pipe: |>

The magrittr pipe has proven so successful and popular that the R core team recently added a "native" pipe operator to base R (version 4.1), denoted `|>`.[1]

- Here's how it works:

```r
mtcars |> subset(cyl == 4) |> head()
mtcars |> subset(cyl == 4) |> (\(x) lm(mpg ~ disp, data = x))()
```

[1] That's actually a `|` followed by a `>`. The default font on these slides just makes it look extra fancy.

# The base R pipe: |>

The magrittr pipe has proven so successful and popular that the R core team recently added a "native" pipe operator to base R (version 4.1), denoted `|>`.[1]

- Here's how it works:

```r
mtcars |> subset(cyl == 4) |> head()
mtcars |> subset(cyl == 4) |> (\(x) lm(mpg ~ disp, data = x))()
```

- This illustrates how the popularity of the tidyverse has repercussions on the development of base R.
- Note that with the native pipe, the RHS function has to be written out together with the brackets (i.e., `... |> head()` instead of `... |> head`).
- Also note the use of the new shorthand inline function syntax, `\(x)`, to pass content to the RHS but not to the first argument.
- Now, should we use the "magrittr" pipe or the native pipe? The native pipe might make more sense in the long term, since it avoids dependencies and might be more efficient. Check out this Stackoverflow post for a discussion of differences.

[1] That's actually a `|` followed by a `>`. The default font on these slides just makes it look extra fancy.

# The tidyverse core developer team

# Data wrangling with dplyr

# Key dplyr verbs

There are five key `dplyr` verbs that you need to learn.[1]

1. `filter()`: Filter (i.e. subset) rows based on their values.

2. `arrange()`: Arrange (i.e. reorder) rows based on their values.

3. `select()`: Select (i.e. subset) columns by their names.

4. `mutate()`: Create new columns.

5. `summarize()`: Collapse multiple rows into a single summary value.[2]

But let's start with studying the key commands using the `starwars` dataset that comes pre-packaged with `dplyr`.

[1] There is much, much more in `dplyr`, and we will look beyond these core functions later. Have a glimpse at the overview at tidyverse.org and at this excellent cheat sheet.

[2] `summarize()` with an "s" works too. I slightly prefer the barbarian version though.

# 1) dplyr::filter()

We can chain multiple filter commands with the pipe (`%>%`), or just separate them within a single filter command using commas.

```
R> starwars %>%
+   filter(
+     species == "Human",
+     height >= 190
+     )
```

```
## # A tibble: 4 × 14
##   name       height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>       <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Darth Va…     202   136 none       white      yellow          41.9 male  mascu…
## 2 Qui-Gon …     193    89 brown      fair       blue            92   male  mascu…
## 3 Dooku         193    80 white      fair       brown          102   male  mascu…
## 4 Bail Pre…     191    NA black      tan        brown           67   male  mascu…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter() *cont.*

Regular expressions work well, too.

```
R> starwars %>%
+   filter(stringr::str_detect(name, "Skywalker"))
```

```
## # A tibble: 3 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
## 1 Luke Sk…     172    77 blond      fair       blue            19   male   mascu…
## 2 Anakin …     188    84 blond      fair       blue            41.9 male   mascu…
## 3 Shmi Sk…     163    NA black      fair       brown           72   female femin…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter() *cont.*

A very common `filter()` use case is identifying (or removing) missing data cases.

```
R> starwars %>%
+   filter(is.na(height))
```

```
## # A tibble: 6 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
## 1 Arvel C…      NA    NA brown      fair       brown             NA male   mascu…
## 2 Finn          NA    NA black      dark       dark              NA male   mascu…
## 3 Rey           NA    NA brown      light      hazel             NA female femin…
## 4 Poe Dam…      NA    NA brown      light      brown             NA male   mascu…
## 5 BB8           NA    NA none       none       black             NA none   mascu…
## 6 Captain…      NA    NA unknown    unknown    unknown           NA <NA>   <NA>
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter() *cont.*

A very common `filter()` use case is identifying (or removing) missing data cases.

```
R> starwars %>%
+   filter(is.na(height))
```

```
## # A tibble: 6 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
## 1 Arvel C…      NA    NA brown      fair       brown             NA male   mascu…
## 2 Finn          NA    NA black      dark       dark              NA male   mascu…
## 3 Rey           NA    NA brown      light      hazel             NA female femin…
## 4 Poe Dam…      NA    NA brown      light      brown             NA male   mascu…
## 5 BB8           NA    NA none       none       black             NA none   mascu…
## 6 Captain…      NA    NA unknown    unknown    unknown           NA <NA>   <NA>
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

To remove missing observations, simply use negation: `filter(!is.na(height))`.

# 1) dplyr::filter() *cont.*

Importantly, when we list several filter conditions, `filter()` interprets them as a Boolean "AND".

```
R> starwars %>%
+   filter(str_detect(name, "Skywalker"),
+                     eye_color == "blue")
```

```
## # A tibble: 2 × 14
##   name        height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>        <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Luke Sky…      172    77 blond      fair       blue            19   male  mascu…
## 2 Anakin S…      188    84 blond      fair       blue            41.9 male  mascu…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# 1) dplyr::filter() *cont.*

Importantly, when we list several filter conditions, `filter()` interprets them as a Boolean "AND".

```
R> starwars %>%
+   filter(str_detect(name, "Skywalker"),
+                     eye_color == "blue")
```

```
## # A tibble: 2 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
## 2 Anakin S…    188    84 blond      fair       blue            41.9 male  mascu…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

We can work with operators `|` ("OR") and `&` ("AND") and combine them with parentheses to specify more complex filter commands, as in:

```
R> starwars %>%
+   filter(species == "Wookiee" | (species == "Human" & height >= 200))
```

# 2) dplyr::arrange()

`arrange()` sorts observations in increasing order by default.

```
R> starwars %>%
+   arrange(birth_year)
```

```
## # A tibble: 87 × 14
##    name      height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Wicket …      88  20   brown      brown      brown              8 male  mascu…
##  2 IG-88        200 140   none       metal      red               15 none  mascu…
##  3 Luke Sk…     172  77   blond      fair       blue              19 male  mascu…
##  4 Leia Or…     150  49   brown      light      brown             19 fema… femin…
##  5 Wedge A…     170  77   brown      fair       hazel             21 male  mascu…
##  6 Plo Koon     188  80   none       orange     black             22 male  mascu…
##  7 Biggs D…     183  84   black      light      brown             24 male  mascu…
##  8 Han Solo     180  80   brown      fair       brown             29 male  mascu…
##  9 Lando C…     177  79   black      dark       brown             31 male  mascu…
## 10 Boba Fe…     183  78.2 black      fair       brown             31.5 male  mascu…
## # … with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

# 2) dplyr::arrange()

`arrange()` sorts observations in increasing order by default.

```
R> starwars %>%
+   arrange(birth_year)
```

```
## # A tibble: 87 × 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Wicket …     88  20   brown      brown      brown            8   male  mascu…
##  2 IG-88       200 140   none       metal      red             15   none  mascu…
##  3 Luke Sk…    172  77   blond      fair       blue            19   male  mascu…
##  4 Leia Or…    150  49   brown      light      brown           19   fema… femin…
##  5 Wedge A…    170  77   brown      fair       hazel           21   male  mascu…
##  6 Plo Koon    188  80   none       orange     black           22   male  mascu…
##  7 Biggs D…    183  84   black      light      brown           24   male  mascu…
##  8 Han Solo    180  80   brown      fair       brown           29   male  mascu…
##  9 Lando C…    177  79   black      dark       brown           31   male  mascu…
## 10 Boba Fe…    183  78.2 black      fair       brown           31.5 male  mascu…
## # … with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

*Note:* Arranging on a character-based column (i.e. strings) will sort alphabetically.

# 2) dplyr::arrange() *cont.*

We can also arrange items in descending order using `arrange(desc())`.

```
R> starwars %>%
+   arrange(desc(birth_year))
```

```
## # A tibble: 87 × 14
##    name     height  mass hair_color skin_color  eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>       <chr>          <dbl> <chr> <chr>
##  1 Yoda         66    17 white      green       brown            896 male  mascu…
##  2 Jabba …     175  1358 <NA>       green-tan,… orange           600 herm… mascu…
##  3 Chewba…     228   112 brown      unknown     blue             200 male  mascu…
##  4 C-3PO       167    75 <NA>       gold        yellow           112 none  mascu…
##  5 Dooku       193    80 white      fair        brown            102 male  mascu…
##  6 Qui-Go…     193    89 brown      fair        blue              92 male  mascu…
##  7 Ki-Adi…     198    82 white      pale        yellow            92 male  mascu…
##  8 Finis …     170    NA blond      fair        blue              91 male  mascu…
##  9 Palpat…     170    75 grey       pale        yellow            82 male  mascu…
## 10 Cliegg…     183    NA brown      fair        blue              82 male  mascu…
## # … with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

# 3) dplyr::select()

Use commas to select multiple columns out of a data frame. (You can also use `<first>:<last>` for consecutive columns). Deselect a column with "-".

```
R> starwars %>%
+   select(name:skin_color, species, -height)
```

```
## # A tibble: 87 × 5
##    name                 mass hair_color    skin_color  species
##    <chr>               <dbl> <chr>         <chr>       <chr>
##  1 Luke Skywalker         77 blond         fair        Human
##  2 C-3PO                  75 <NA>          gold        Droid
##  3 R2-D2                  32 <NA>          white, blue Droid
##  4 Darth Vader           136 none          white       Human
##  5 Leia Organa            49 brown         light       Human
##  6 Owen Lars             120 brown, grey   light       Human
##  7 Beru Whitesun lars     75 brown         light       Human
##  8 R5-D4                  32 <NA>          white, red  Droid
##  9 Biggs Darklighter      84 black         light       Human
## 10 Obi-Wan Kenobi         77 auburn, white fair        Human
## # … with 77 more rows
```

# 3) dplyr::select() *cont.*

You can also rename some (or all) of your selected variables in place.

```
R> starwars %>%
+   select(alias = name, crib = homeworld, sex = gender)
```

```
## # A tibble: 87 × 3
##    alias              crib     sex
##    <chr>              <chr>    <chr>
##  1 Luke Skywalker     Tatooine masculine
##  2 C-3PO              Tatooine masculine
##  3 R2-D2              Naboo    masculine
##  4 Darth Vader        Tatooine masculine
##  5 Leia Organa        Alderaan feminine
##  6 Owen Lars          Tatooine masculine
##  7 Beru Whitesun lars Tatooine feminine
##  8 R5-D4              Tatooine masculine
##  9 Biggs Darklighter  Tatooine masculine
## 10 Obi-Wan Kenobi     Stewjon  masculine
## # … with 77 more rows
```

# 3) dplyr::select() *cont.*

You can also rename some (or all) of your selected variables in place.

```
R> starwars %>%
+   select(alias = name, crib = homeworld, sex = gender)
```

```
## # A tibble: 87 × 3
##    alias              crib     sex
##    <chr>              <chr>    <chr>
##  1 Luke Skywalker     Tatooine masculine
##  2 C-3PO              Tatooine masculine
##  3 R2-D2              Naboo    masculine
##  4 Darth Vader        Tatooine masculine
##  5 Leia Organa        Alderaan feminine
##  6 Owen Lars          Tatooine masculine
##  7 Beru Whitesun lars Tatooine feminine
##  8 R5-D4              Tatooine masculine
##  9 Biggs Darklighter  Tatooine masculine
## 10 Obi-Wan Kenobi     Stewjon  masculine
## # … with 77 more rows
```

If you just want to rename columns without subsetting them, you can use `rename()`.

# 3) dplyr::select() *cont.*

The `select(contains(<PATTERN>))` option provides a nice shortcut in relevant cases.

```
R> starwars %>%
+   select(name, contains("color"))
```

```
## # A tibble: 87 × 4
##    name               hair_color    skin_color  eye_color
##    <chr>              <chr>         <chr>       <chr>
##  1 Luke Skywalker     blond         fair        blue
##  2 C-3PO              <NA>          gold        yellow
##  3 R2-D2              <NA>          white, blue red
##  4 Darth Vader        none          white       yellow
##  5 Leia Organa        brown         light       brown
##  6 Owen Lars          brown, grey   light       blue
##  7 Beru Whitesun lars brown         light       blue
##  8 R5-D4              <NA>          white, red  red
##  9 Biggs Darklighter  black         light       brown
## 10 Obi-Wan Kenobi     auburn, white fair        blue-gray
## # … with 77 more rows
```

# 3) dplyr::select() *cont.*

The `select(contains(<PATTERN>))` option provides a nice shortcut in relevant cases.

```
R> starwars %>%
+   select(name, contains("color"))
```

```
## # A tibble: 87 × 4
##    name               hair_color    skin_color  eye_color
##    <chr>              <chr>         <chr>       <chr>
##  1 Luke Skywalker     blond         fair        blue
##  2 C-3PO              <NA>          gold        yellow
##  3 R2-D2              <NA>          white, blue red
##  4 Darth Vader        none          white       yellow
##  5 Leia Organa        brown         light       brown
##  6 Owen Lars          brown, grey   light       blue
##  7 Beru Whitesun lars brown         light       blue
##  8 R5-D4              <NA>          white, red  red
##  9 Biggs Darklighter  black         light       brown
## 10 Obi-Wan Kenobi     auburn, white fair        blue-gray
## # … with 77 more rows
```

There are many more useful selection helpers, such as `starts_with()`, `ends_with()`, and `matches()`. See here for an overview.

# 3) dplyr::select() *cont.*

The `select( ... , everything())` option is another useful shortcut if you only want to bring some variable(s) to the "front" of a data frame.

```
R> starwars %>%
+   select(species, homeworld, everything()) %>%
+   head(5)
```

```
## # A tibble: 5 × 14
##   species homeworld name           height  mass hair_color skin_color  eye_color
##   <chr>   <chr>     <chr>           <int> <dbl> <chr>      <chr>       <chr>
## 1 Human   Tatooine  Luke Skywalker    172    77 blond      fair        blue
## 2 Droid   Tatooine  C-3PO             167    75 <NA>       gold        yellow
## 3 Droid   Naboo     R2-D2              96    32 <NA>       white, blue red
## 4 Human   Tatooine  Darth Vader       202   136 none       white       yellow
## 5 Human   Alderaan  Leia Organa       150    49 brown      light       brown
## # … with 6 more variables: birth_year <dbl>, sex <chr>, gender <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

# 3) dplyr::select() *cont.*

The `select( ... , everything())` option is another useful shortcut if you only want to bring some variable(s) to the "front" of a data frame.

```
R> starwars %>%
+   select(species, homeworld, everything()) %>%
+   head(5)
```

```
## # A tibble: 5 × 14
##   species homeworld name           height  mass hair_color skin_color  eye_color
##   <chr>   <chr>     <chr>           <int> <dbl> <chr>      <chr>       <chr>
## 1 Human   Tatooine  Luke Skywalker    172    77 blond      fair        blue
## 2 Droid   Tatooine  C-3PO             167    75 <NA>       gold        yellow
## 3 Droid   Naboo     R2-D2              96    32 <NA>       white, blue red
## 4 Human   Tatooine  Darth Vader       202   136 none       white       yellow
## 5 Human   Alderaan  Leia Organa       150    49 brown      light       brown
## # … with 6 more variables: birth_year <dbl>, sex <chr>, gender <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

*Note:* The new `relocate()` function available in dplyr 1.0.0 has brought a lot more functionality to the ordering of columns. See here.

# 4) dplyr::mutate()

You can create new columns from scratch with `mutate()`, or (more commonly) as transformations of existing columns.

```
R> starwars %>%
+   select(name, birth_year) %>%
+   mutate(
+     dog_years = birth_year * 7, ## Separate with a comma
+     comment = paste0(name, " is ", dog_years, " in dog years.")
+     ) %>%
+   slice(1:6) # Just show first six observations
```

```
## # A tibble: 6 × 4
##   name           birth_year dog_years comment
##   <chr>               <dbl>     <dbl> <chr>
## 1 Luke Skywalker       19        133  Luke Skywalker is 133 in dog years.
## 2 C-3PO               112        784  C-3PO is 784 in dog years.
## 3 R2-D2                33        231  R2-D2 is 231 in dog years.
## 4 Darth Vader          41.9      293. Darth Vader is 293.3 in dog years.
## 5 Leia Organa          19        133  Leia Organa is 133 in dog years.
## 6 Owen Lars            52        364  Owen Lars is 364 in dog years.
```

# 4) dplyr::mutate()

You can create new columns from scratch with `mutate()`, or (more commonly) as transformations of existing columns.

```
R> starwars %>%
+   select(name, birth_year) %>%
+   mutate(
+     dog_years = birth_year * 7, ## Separate with a comma
+     comment = paste0(name, " is ", dog_years, " in dog years.")
+     ) %>%
+   slice(1:6) # Just show first six observations
```

```
## # A tibble: 6 × 4
##   name           birth_year dog_years comment
##   <chr>               <dbl>     <dbl> <chr>
## 1 Luke Skywalker       19         133  Luke Skywalker is 133 in dog years.
## 2 C-3PO               112         784  C-3PO is 784 in dog years.
## 3 R2-D2                33         231  R2-D2 is 231 in dog years.
## 4 Darth Vader          41.9       293. Darth Vader is 293.3 in dog years.
## 5 Leia Organa          19         133  Leia Organa is 133 in dog years.
## 6 Owen Lars            52         364  Owen Lars is 364 in dog years.
```

*Note:* `mutate()` is order aware. So you can chain multiple mutates in a single call.

# 4) dplyr::mutate() *cont.*

Boolean, logical and conditional operators all work well with `mutate()` too.

```
R> starwars %>%
+   select(name, height) %>%
+   filter(name %in% c("Luke Skywalker", "Anakin Skywalker")) %>%
+   mutate(tall1 = height > 180) %>%
+   mutate(tall2 = ifelse(height > 180, "Tall", "Short")) ## Same effect, but can choose labels
```

```
## # A tibble: 2 × 4
##   name             height tall1 tall2
##   <chr>             <int> <lgl> <chr>
## 1 Luke Skywalker      172 FALSE Short
## 2 Anakin Skywalker    188 TRUE  Tall
```

# 4) dplyr::mutate() *cont.*

Lastly, combining `mutate()` with the `across()` feature allows you to easily work on a subset of variables. For example:

```
R> starwars %>%
+   select(name:eye_color) %>%
+   mutate(across(where(is.character), toupper)) %>%
+   head(5)
```

```
## # A tibble: 5 × 6
##   name           height  mass hair_color skin_color  eye_color
##   <chr>           <int> <dbl> <chr>      <chr>       <chr>
## 1 LUKE SKYWALKER    172    77 BLOND      FAIR        BLUE
## 2 C-3PO             167    75 <NA>       GOLD        YELLOW
## 3 R2-D2              96    32 <NA>       WHITE, BLUE RED
## 4 DARTH VADER       202   136 NONE       WHITE       YELLOW
## 5 LEIA ORGANA       150    49 BROWN      LIGHT       BROWN
```

# 4) dplyr::mutate() *cont.*

Lastly, combining `mutate()` with the `across()` feature allows you to easily work on a subset of variables. For example:

```
R> starwars %>%
+   select(name:eye_color) %>%
+   mutate(across(where(is.character), toupper)) %>%
+   head(5)
```

```
## # A tibble: 5 × 6
##   name           height  mass hair_color skin_color  eye_color
##   <chr>           <int> <dbl> <chr>      <chr>       <chr>
## 1 LUKE SKYWALKER    172    77 BLOND      FAIR        BLUE
## 2 C-3PO             167    75 <NA>       GOLD        YELLOW
## 3 R2-D2              96    32 <NA>       WHITE, BLUE RED
## 4 DARTH VADER       202   136 NONE       WHITE       YELLOW
## 5 LEIA ORGANA       150    49 BROWN      LIGHT       BROWN
```

*Note:* More on `across()` and `where()` later!

# 5) dplyr::summarize()

You can summarize variables with all sorts of operations (e.g., `mean()`, `median()`, `n()`, `n_distinct()`, `sum()`, `first()`, `last()`, ...).

```
R> starwars %>%
+   group_by(species, gender) %>%
+   summarize(mean_height = mean(height, na.rm = TRUE)) %>%
+   head(5)
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups` argument.
```

```
## # A tibble: 5 × 3
## # Groups:   species [5]
##   species  gender    mean_height
##   <chr>    <chr>           <dbl>
## 1 Aleena   masculine          79
## 2 Besalisk masculine         198
## 3 Cerean   masculine         198
## 4 Chagrian masculine         196
## 5 Clawdite feminine          168
```

# 5) dplyr::summarize()

You can summarize variables with all sorts of operations (e.g., `mean()`, `median()`, `n()`, `n_distinct()`, `sum()`, `first()`, `last()`, ...).

```
R> starwars %>%
+   group_by(species, gender) %>%
+   summarize(mean_height = mean(height, na.rm = TRUE)) %>%
+   head(5)
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups` argument.
```

```
## # A tibble: 5 × 3
## # Groups:   species [5]
##   species  gender    mean_height
##   <chr>    <chr>           <dbl>
## 1 Aleena   masculine          79
## 2 Besalisk masculine         198
## 3 Cerean   masculine         198
## 4 Chagrian masculine         196
## 5 Clawdite feminine          168
```

*Note:* This is particularly useful in combination with the `group_by()` command. Again, more on this later!

# 5) dplyr::summarize() *cont.*

Note that including `na.rm = TRUE` is usually a good idea with the functions fed into `summarize()` Otherwise, any missing value will propagate to the summarized value too.

```
R> ## Probably not what we want
R> starwars %>%
+   summarize(mean_height = mean(height))
```

```
## # A tibble: 1 × 1
##   mean_height
##         <dbl>
## 1          NA
```

```
R> ## Much better
R> starwars %>%
+   summarize(mean_height = mean(height, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   mean_height
##         <dbl>
## 1        174.
```

# 5) dplyr::summarize() *cont.*

The same `across()`-based workflow that we saw with `mutate()` a few slides back also works with `summarize()`. For example:

```
R> starwars %>%
+   group_by(species) %>%
+   summarize(across(where(is.numeric), mean, na.rm = TRUE)) %>%
+   head(5)
```

```
## # A tibble: 5 × 4
##   species  height  mass birth_year
##   <chr>     <dbl> <dbl>      <dbl>
## 1 Aleena       79    15        NaN
## 2 Besalisk    198   102        NaN
## 3 Cerean      198    82         92
## 4 Chagrian    196   NaN        NaN
## 5 Clawdite    168    55        NaN
```

# Grouping with dplyr::group_by()

With `group_by()`, you can create a "grouped" copy of a table grouped by unique values of a column. If multiple columns are specified, the function groups by all available combinations of values.

```
R> by_species_gender ← starwars %>% group_by(species, gender)
R> by_species_gender
```

```
## # A tibble: 87 × 14
## # Groups:   species, gender [42]
##    name     height  mass hair_color  skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke S…     172    77 blond       fair       blue            19   male  mascu…
##  2 C-3PO       167    75 <NA>        gold       yellow         112   none  mascu…
##  3 R2-D2        96    32 <NA>        white, bl… red             33   none  mascu…
##  4 Darth …     202   136 none        white      yellow          41.9 male  mascu…
##  5 Leia O…     150    49 brown       light      brown           19   fema… femin…
##  6 Owen L…     178   120 brown, grey light      blue            52   male  mascu…
##  7 Beru W…     165    75 brown       light      blue            47   fema… femin…
##  8 R5-D4        97    32 <NA>        white, red red             NA   none  mascu…
##  9 Biggs …     183    84 black       light      brown           24   male  mascu…
## 10 Obi-Wa…     182    77 auburn, wh… fair       blue-gray       57   male  mascu…
## # … with 77 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

# Grouping with dplyr::group_by() *cont.*

## More notes on grouping

- Grouping doesn't change how the data looks (apart from listing how it's grouped).
- Grouping changes how it acts with other dplyr verbs such as `summarize()` and `mutate()`, as we've already seen.
- By default, `group_by()` overrides existing grouping. Use `.add = TRUE` to append instead.
- By default, groups formed by factor levels that don't appear in the data are dropped. Set `.drop = FALSE` if you want to keep them.
- `ungroup()` removes existing grouping.
- `dplyr` notifies you about grouping variables every time you do operations on or with them. If you find these messages annoying, switch them off with `options(dplyr.summarise.inform = FALSE)`.

```
R> options(dplyr.summarise.inform = FALSE)
R> by_species_gender %>%
+   summarize(mean(height, na.rm = TRUE)) %>%
+   filter(n_distinct(gender) ==2)
```

```
## # A tibble: 8 × 3
## # Groups:   species [4]
##   species  gender    `mean(height, na.rm = TRUE)`
##   <chr>    <chr>                            <dbl>
## 1 Droid    feminine                          96
## 2 Droid    masculine                        140
## 3 Human    feminine                         160.
## 4 Human    masculine                        182.
## 5 Kaminoan feminine                         213
## 6 Kaminoan masculine                        229
## 7 Twi'lek  feminine                         178
## 8 Twi'lek  masculine                        180
```

# Other dplyr goodies

# Other dplyr goodies

`slice()`: Subset rows by position rather than filtering by values. There's also `slice_sample()` to randomly select rows, `slice_head()` and `slice_tail()` to select first or last rows, and more.

```
R> starwars %>% slice(c(1, 5))
```

```
## # A tibble: 2 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
## 1 Luke Sk…     172    77 blond      fair       blue              19 male   mascu…
## 2 Leia Or…     150    49 brown      light      brown             19 female femin…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

# Other dplyr goodies

`slice()`: Subset rows by position rather than filtering by values. There's also `slice_sample()` to randomly select rows, `slice_head()` and `slice_tail()` to select first or last rows, and more.

```
R> starwars %>% slice(c(1, 5))
```

```
## # A tibble: 2 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>  <chr>
## 1 Luke Sk…     172    77 blond      fair       blue              19 male   mascu…
## 2 Leia Or…     150    49 brown      light      brown             19 female femin…
## # … with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

`pull()`: Extract a column from as a data frame as a vector or scalar.

```
R> starwars %>% filter(gender=="feminine") %>% pull(height)
```

```
##  [1] 150 165 150 163 178 184 157 170 166 165 168 213 167  96 178  NA 165
```

# Other dplyr goodies *cont.*

`count()` and `distinct()`: Number and isolate unique observations.

```
R> starwars %>% count(species) %>% head(6)
```

```
## # A tibble: 6 × 2
##   species      n
##   <chr>    <int>
## 1 Aleena       1
## 2 Besalisk     1
## 3 Cerean       1
## 4 Chagrian     1
## 5 Clawdite     1
## 6 Droid        6
```

```
R> starwars %>% distinct(species) %>% pull() %>% sort() %>% magrittr::extract(1:5)
```

```
## [1] "Aleena"   "Besalisk" "Cerean"   "Chagrian" "Clawdite"
```

# Other dplyr goodies *cont.*

`count()` and `distinct()`: Number and isolate unique observations.

```
R> starwars %>% count(species) %>% head(6)
```

```
## # A tibble: 6 × 2
##   species      n
##   <chr>    <int>
## 1 Aleena       1
## 2 Besalisk     1
## 3 Cerean       1
## 4 Chagrian     1
## 5 Clawdite     1
## 6 Droid        6
```

```
R> starwars %>% distinct(species) %>% pull() %>% sort() %>% magrittr::extract(1:5)
```

```
## [1] "Aleena"   "Besalisk" "Cerean"   "Chagrian" "Clawdite"
```

You could also use a combination of `mutate()`, `group_by()`, and `n()`, e.g. `starwars %>% group_by(species) %>% mutate(num = n())`.

# Other dplyr goodies *cont.*

`where()`: Select the variables for which a function returns true.

```
R> starwars %>% select(where(is.numeric)) %>% names()
```

```
## [1] "height"     "mass"       "birth_year"
```

# Other dplyr goodies *cont.*

`where()`: Select the variables for which a function returns true.

```
R> starwars %>% select(where(is.numeric)) %>% names()
```

```
## [1] "height"     "mass"       "birth_year"
```

`across()`: Summarize or mutate multiple variables in the same way. More information here.

```
R> starwars %>%
+   mutate(across(where(is.numeric), scale)) %>%
+   head(3)
```

```
## # A tibble: 3 × 14
##   name  height[,1] mass[,1] hair_color skin_color eye_color birth_year[,1] sex
##   <chr>      <dbl>    <dbl> <chr>      <chr>      <chr>              <dbl> <chr>
## 1 Luke…    -0.0678   -0.120 blond      fair       blue              -0.443 male
## 2 C-3PO    -0.212    -0.132 <NA>       gold       yellow             0.158 none
## 3 R2-D2    -2.25     -0.385 <NA>       white, bl… red               -0.353 none
## # … with 6 more variables: gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

# Other dplyr goodies *cont.*

`case_when()`: Vectorize multiple `if_else()` (or base R `ifelse()`) statements.

```
R> starwars %>%
+   mutate(
+     height_cat = case_when(
+       height < 160 ~ "tiny",
+       height >= 160 & height < 190 ~ "medium",
+       height >= 190 & height < 220 ~ "tall",
+       height >= 220 ~ "giant"
+     )
+   ) %>%
+   pull(height_cat) %>% table()
```

```
## .
##  giant medium   tall   tiny
##      5     45     18     13
```

# Other dplyr goodies *cont.*

`case_when()`: Vectorize multiple `if_else()` (or base R `ifelse()`) statements.

```
R> starwars %>%
+   mutate(
+     height_cat = case_when(
+       height < 160 ~ "tiny",
+       height >= 160 & height < 190 ~ "medium",
+       height >= 190 & height < 220 ~ "tall",
+       height >= 220 ~ "giant"
+     )
+   ) %>%
+   pull(height_cat) %>% table()
```

```
## .
##  giant medium   tall   tiny
##      5     45     18     13
```

There are also a whole class of window functions for getting leads and lags, ranking, creating cumulative aggregates, etc. See `vignette("window-functions")`.

# Other dplyr goodies *cont.*

`case_when()`: Vectorize multiple `if_else()` (or base R `ifelse()`) statements.

```
R> starwars %>%
+   mutate(
+     height_cat = case_when(
+       height < 160 ~ "tiny",
+       height >= 160 & height < 190 ~ "medium",
+       height >= 190 & height < 220 ~ "tall",
+       height >= 220 ~ "giant"
+     )
+   ) %>%
+   pull(height_cat) %>% table()
```

```
## .
##  giant medium   tall   tiny
##      5     45     18     13
```

There are also a whole class of window functions for getting leads and lags, ranking, creating cumulative aggregates, etc. See `vignette("window-functions")`.

`inner_join()`, `left_join()`, `right_join()`: Enough already, we'll talk about this in the next session!

# Data tidying with tidyr

# Key tidyr verbs

`tidyr` is part of the core tidyverse. There are four key `tidyr` verbs that you need to learn.

1. `pivot_longer()`: Pivot wide data into long format (i.e. "melt").[1]
2. `pivot_wider()`: Pivot long data into wide format (i.e. "cast").[2]
3. `separate()`: Separate (i.e. split) one column into multiple columns.
4. `unite()`: Unite (i.e. combine) multiple columns into one.

[1] Updated version of `tidyr::gather()`.

[2] Updated version of `tidyr::spread()`.

# On "longer" and "wider" datasets

Remember the **key philosophy for tidy data**?

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

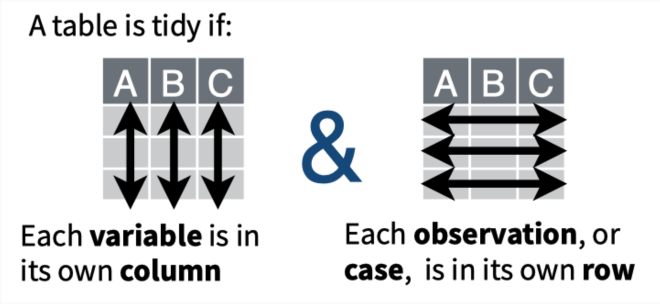
One of the most common tasks for data scientists is to **reshape** data from one form to the other.

There are **multiple ways to store the same data in a dataset** (or across multiple tables; but more on that in the next session).

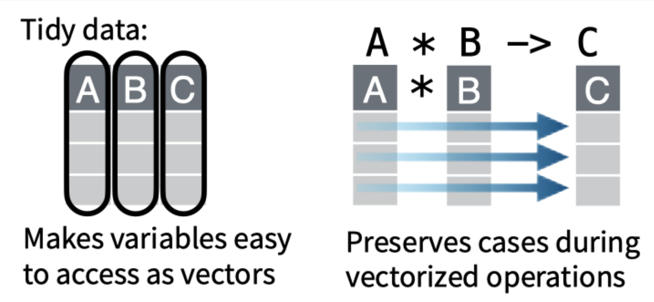
Here, we learn how to shift between

- **"wider"** formats, i.e. data being stored across more columns and
- **"longer"** formats, i.e. data being stored across more rows.

Tidy data in a nutshell




Benefits of tidy data

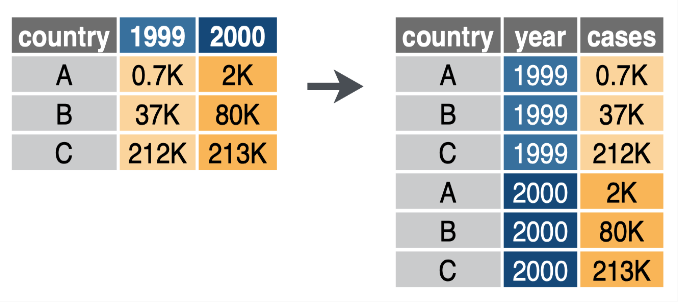


# From wide to long to wide

## From wider to longer

- `pivot_longer()` pivots `cols` columns, moving column names into a `names_to` column, and column values into a `values_to` column.
- Recall a panel study design with multiple observations per unit.
- In the classical long format, each row represents one observation.
- Note how this is approaching the ideal of **tidy data**.

`pivot_longer()`



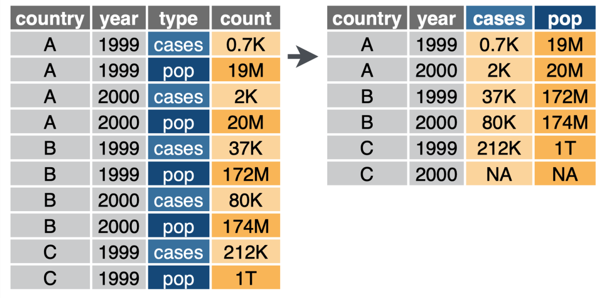
| country | 1999 | 2000 |
|---|---|---|
| A | 0.7K | 2K |
| B | 37K | 80K |
| C | 212K | 213K |

| country | year | cases |
|---|---|---|
| A | 1999 | 0.7K |
| B | 1999 | 37K |
| C | 1999 | 212K |
| A | 2000 | 2K |
| B | 2000 | 80K |
| C | 2000 | 213K |

## From longer to wider

- `pivot_wider()` pivots a `names_from` and a `values_from` column into a rectangular field of cells.
- In a panel study design, this would allow you to have one variable per measurement (e.g., pre- and posttreatment outcome variable).
- While this is nice for the human eye, it is sometimes not what fits the tidyverse workflow. Also, wenn you have multiple repeated measurements (think: variables in a population survey), the number of columns is quickly inflated. Be ready to `pivot_longer()`.

`pivot_wider()`



| country | year | type | count |
|---|---|---|---|
| A | 1999 | cases | 0.7K |
| A | 1999 | pop | 19M |
| A | 2000 | cases | 2K |
| A | 2000 | pop | 20M |
| B | 1999 | cases | 37K |
| B | 1999 | pop | 172M |
| B | 2000 | cases | 80K |
| B | 2000 | pop | 174M |
| C | 1999 | cases | 212K |
| C | 1999 | pop | 1T |

| country | year | cases | pop |
|---|---|---|---|
| A | 1999 | 0.7K | 19M |
| A | 2000 | 2K | 20M |
| B | 1999 | 37K | 172M |
| B | 2000 | 80K | 174M |
| C | 1999 | 212K | 1T |
| C | 2000 | NA | NA |

# 1) tidyr::pivot_longer()

```
R> stocks = data.frame( ## Could use "tibble" instead of "data.frame" if you prefer
+   time = as.Date('2009-01-01') + 0:1,
+   X = rnorm(2, 0, 1),
+   Y = rnorm(2, 0, 2),
+   Z = rnorm(2, 0, 4)
+   )
R> stocks
```

```
##          time          X          Y         Z
## 1 2009-01-01  0.1890718 -0.5036369 -5.172738
## 2 2009-01-02 -0.1800420  0.2868808  1.193378
```

# 1) tidyr::pivot_longer()

```
R> stocks = data.frame( ## Could use "tibble" instead of "data.frame" if you prefer
+   time = as.Date('2009-01-01') + 0:1,
+   X = rnorm(2, 0, 1),
+   Y = rnorm(2, 0, 2),
+   Z = rnorm(2, 0, 4)
+   )
R> stocks
```

```
##         time          X          Y         Z
## 1 2009-01-01  0.1890718 -0.5036369 -5.172738
## 2 2009-01-02 -0.1800420  0.2868808  1.193378
```

```
R> tidy_stocks ← stocks %>% pivot_longer(-time, names_to="stock", values_to="price")
R> tidy_stocks
```

```
## # A tibble: 6 × 3
##   time       stock  price
##   <date>     <chr>  <dbl>
## 1 2009-01-01 X      0.189
## 2 2009-01-01 Y     -0.504
## 3 2009-01-01 Z     -5.17
## 4 2009-01-02 X     -0.180
## 5 2009-01-02 Y      0.287
```

# 2) tidyr::pivot_wider()

```
R> tidy_stocks %>% pivot_wider(names_from = stock, values_from = price)
```

```
## # A tibble: 2 × 4
##   time             X      Y     Z
##   <date>       <dbl>  <dbl> <dbl>
## 1 2009-01-01  0.189 -0.504 -5.17
## 2 2009-01-02 -0.180  0.287  1.19
```

# 2) tidyr::pivot_wider()

```
R> tidy_stocks %>% pivot_wider(names_from = stock, values_from = price)
```

```
## # A tibble: 2 × 4
##   time             X      Y     Z
##   <date>       <dbl>  <dbl> <dbl>
## 1 2009-01-01  0.189 -0.504 -5.17
## 2 2009-01-02 -0.180  0.287  1.19
```

```
R> tidy_stocks %>% pivot_wider(names_from= time, values_from = price)
```

```
## # A tibble: 3 × 3
##   stock `2009-01-01` `2009-01-02`
##   <chr>        <dbl>        <dbl>
## 1 X            0.189       -0.180
## 2 Y           -0.504        0.287
## 3 Z           -5.17         1.19
```

# 2) tidyr::pivot_wider()

```
R> tidy_stocks %>% pivot_wider(names_from = stock, values_from = price)
```

```
## # A tibble: 2 × 4
##   time            X      Y     Z
##   <date>      <dbl>  <dbl> <dbl>
## 1 2009-01-01  0.189 -0.504 -5.17
## 2 2009-01-02 -0.180  0.287  1.19
```

```
R> tidy_stocks %>% pivot_wider(names_from= time, values_from = price)
```

```
## # A tibble: 3 × 3
##   stock `2009-01-01` `2009-01-02`
##   <chr>        <dbl>        <dbl>
## 1 X            0.189       -0.180
## 2 Y           -0.504        0.287
## 3 Z           -5.17         1.19
```

*Note*: The second example — which has combined different pivoting arguments — has effectively transposed the data.

# 3) tidyr::separate()

Sometimes, cell values provide information that should be stored in separate columns. `separate()` offers one way of doing this. (*Side note*: Once you learn regular expressions, you will have an even more powerful tool for this task.)

```
R> economists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
R> economists
```

```
##              name
## 1      Adam.Smith
## 2  Paul.Samuelson
## 3 Milton.Friedman
```

# 3) tidyr::separate()

Sometimes, cell values provide information that should be stored in separate columns. `separate()` offers one way of doing this. (*Side note*: Once you learn regular expressions, you will have an even more powerful tool for this task.)

```
R> economists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
R> economists
```

```
##              name
## 1      Adam.Smith
## 2  Paul.Samuelson
## 3 Milton.Friedman
```

`separate()` in action:

```
R> economists %>% separate(name, c("first_name", "last_name"))
```

```
##   first_name last_name
## 1       Adam     Smith
## 2       Paul Samuelson
## 3     Milton  Friedman
```

# 3) tidyr::separate()

Sometimes, cell values provide information that should be stored in separate columns. `separate()` offers one way of doing this. (*Side note*: Once you learn regular expressions, you will have an even more powerful tool for this task.)

```
R> economists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
R> economists
```

```
##              name
## 1      Adam.Smith
## 2  Paul.Samuelson
## 3 Milton.Friedman
```

`separate()` in action:

```
R> economists %>% separate(name, c("first_name", "last_name"))
```

```
##   first_name last_name
## 1       Adam     Smith
## 2       Paul Samuelson
## 3     Milton  Friedman
```

You can also specify the separation character with `separate( ... , sep=".")`. The way `sep` works also depends on column typ (character vs. numberic). Check out the function reference.

# 3) tidyr::separate() *cont.*

A related function is `separate_rows()`, for splitting up cells that contain multiple fields or observations (a frustratingly common occurence with survey data).

```
R> jobs = data.frame(
+   name = c("Jack", "Jill"),
+   occupation = c("Homemaker", "Philosopher, Philanthropist, Troublemaker")
+   )
R> jobs
```

```
##   name                                occupation
## 1 Jack                                 Homemaker
## 2 Jill Philosopher, Philanthropist, Troublemaker
```

# 3) tidyr::separate() *cont.*

A related function is `separate_rows()`, for splitting up cells that contain multiple fields or observations (a frustratingly common occurence with survey data).

```
R> jobs = data.frame(
+   name = c("Jack", "Jill"),
+   occupation = c("Homemaker", "Philosopher, Philanthropist, Troublemaker")
+   )
R> jobs
```

```
##   name                                occupation
## 1 Jack                                 Homemaker
## 2 Jill Philosopher, Philanthropist, Troublemaker
```

`separate_rows()` in action:

```
R> jobs %>% separate_rows(occupation)
```

```
## # A tibble: 4 × 2
##   name  occupation
##   <chr> <chr>
## 1 Jack  Homemaker
## 2 Jill  Philosopher
```

# 4) tidyr::unite()

`separate()` has a complementary function, `unite()`. Unsurprinsingly, it unites values from multiple columns into one.

```
R> gdp = data.frame(
+   yr = rep(2016, times = 3),
+   mnth = rep(1, times = 3),
+   dy = 1:3,
+   gdp = rnorm(3, mean = 100, sd = 2)
+   )
R> gdp
```

```
##     yr mnth dy       gdp
## 1 2016    1  1  98.81436
## 2 2016    1  2  97.73040
## 3 2016    1  3 101.38806
```

```
R> ## Combine "yr", "mnth", and "dy" into one "date" column
R> gdp %>% unite(date, c("yr", "mnth", "dy"), sep = "-")
```

```
##       date       gdp
## 1 2016-1-1  98.81436
## 2 2016-1-2  97.73040
## 3 2016-1-3 101.38806
```

# 4) tidyr::unite() *cont.*

Note that `unite()` will automatically create a character variable. You can see this better if we convert it to a tibble.

```
R> gdp_u = gdp %>%
+   unite(date,
+         c("yr", "mnth", "dy"),
+         sep = "-") %>%
+   as_tibble()
R> gdp_u
```

```
## # A tibble: 3 × 2
##   date       gdp
##   <chr>    <dbl>
## 1 2016-1-1  98.8
## 2 2016-1-2  97.7
## 3 2016-1-3 101.
```

# 4) tidyr::unite() *cont.*

Note that `unite()` will automatically create a character variable. You can see this better if we convert it to a tibble.

```
R> gdp_u = gdp %>%
+   unite(date,
+         c("yr", "mnth", "dy"),
+         sep = "-") %>%
+   as_tibble()
R> gdp_u
```

```
## # A tibble: 3 × 2
##   date       gdp
##   <chr>    <dbl>
## 1 2016-1-1  98.8
## 2 2016-1-2  97.7
## 3 2016-1-3 101.
```

If you want to convert it to something else (e.g. date or numeric) then you will need to modify it using `mutate()`. See below for an example, using the lubridate package's super helpful date conversion functions.

```
R> library(lubridate)
R> gdp_u %>% mutate(date = ymd(date))
```

```
## # A tibble: 3 × 2
##   date         gdp
##   <date>     <dbl>
## 1 2016-01-01  98.8
## 2 2016-01-02  97.7
## 3 2016-01-03 101.
```

# Other tidyr goodies

`crossing()`: Get the full combination of a group of variables.[1]

```
R> crossing(side=c("left", "right"), height=c("top", "bottom"))
```

```
## # A tibble: 4 × 2
##   side  height
##   <chr> <chr>
## 1 left  bottom
## 2 left  top
## 3 right bottom
## 4 right top
```

[1] See `?expand()` and `?complete()` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames. Base R alternative: `expand.grid()`.

# Other tidyr goodies

`crossing()`: Get the full combination of a group of variables.[1]

```
R> crossing(side=c("left", "right"), height=c("top", "bottom"))
```

```
## # A tibble: 4 × 2
##   side  height
##   <chr> <chr>
## 1 left  bottom
## 2 left  top
## 3 right bottom
## 4 right top
```

`drop_na(data, ...)`: Drop rows containing NAs in `...` columns.

[1] See `?expand()` and `?complete()` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames. Base R alternative: `expand.grid()`.

# Other tidyr goodies

`crossing()`: Get the full combination of a group of variables.[1]

```
R> crossing(side=c("left", "right"), height=c("top", "bottom"))
```

```
## # A tibble: 4 × 2
##   side  height
##   <chr> <chr>
## 1 left  bottom
## 2 left  top
## 3 right bottom
## 4 right top
```

`drop_na(data, ... )`: Drop rows containing NAs in `...` columns.

`fill(data, ... , direction = c("down", "up"))`: Fill in NAs in `...` columns with most recent non-NA values.

[1] See `?expand()` and `?complete()` for more specialized functions that allow you to fill in (implicit) missing data or variable combinations in existing data frames. Base R alternative: `expand.grid()`.

# Tidy programming

# Tidy programming basics

"Tidy programming" is not a strictly defined practice in the tidyverse. However, there are some common programming strategies that help you keep your code and workflow tidy. These include:

- Pipes (you already know that ✅)
- User-generated functions
- Functional programming with `purrr`

# Tidy programming basics

"Tidy programming" is not a strictly defined practice in the tidyverse. However, there are some common programming strategies that help you keep your code and workflow tidy. These include:

- Pipes (you already know that ✅)
- User-generated functions
- Functional programming with `purrr`

The latter two are extremely helpful - in particular when you are confronted with iterative tasks.

# Tidy programming basics

"Tidy programming" is not a strictly defined practice in the tidyverse. However, there are some common programming strategies that help you keep your code and workflow tidy. These include:

- Pipes (you already know that ✅)
- User-generated functions
- Functional programming with `purrr`

The latter two are extremely helpful - in particular when you are confronted with iterative tasks.

We will now learn the basics of creating your own functions and functional programming with R. There is much more to learn about these topics, so we will revisit them as the course progresses.

# Creating functions

## Why creating functions?

That's a legit question. There are 18,000+ **packages** on CRAN (and many, many more on GitHub and other repositories) containing zillions of functions. Why should you create yet another one?

- Every data science project is unique. There are problems only you have to solve.
- For problems that are repetitive, you'll quickly look for options to automate the task.
- Functions are a great way to automate.

# Creating functions

## Why creating functions?

That's a legit question. There are 18,000+ **packages** on CRAN (and many, many more on GitHub and other repositories) containing zillions of functions. Why should you create yet another one?

- Every data science project is unique. There are problems only you have to solve.
- For problems that are repetitive, you'll quickly look for options to automate the task.
- Functions are a great way to automate.

## Examples where creating functions makes sense

# Creating functions

## Why creating functions?

That's a legit question. There are 18,000+ **packages** on CRAN (and many, many more on GitHub and other repositories) containing zillions of functions. Why should you create yet another one?

- Every data science project is unique. There are problems only you have to solve.
- For problems that are repetitive, you'll quickly look for options to automate the task.
- Functions are a great way to automate.

## Examples where creating functions makes sense

1. You want to scrape thousands of websites. This implies multiple steps, from downloading to parsing and cleaning. All these steps can be achieved with existing functions, but the fine-tuning is specific to the set of websites. You build one (or a set of) scraping functions that take the websites as input and return a cleaned data frame ready to be analyzed.

# Creating functions

## Why creating functions?

That's a legit question. There are 18,000+ **packages** on CRAN (and many, many more on GitHub and other repositories) containing zillions of functions. Why should you create yet another one?

- Every data science project is unique. There are problems only you have to solve.
- For problems that are repetitive, you'll quickly look for options to automate the task.
- Functions are a great way to automate.

## Examples where creating functions makes sense

1. You want to scrape thousands of websites. This implies multiple steps, from downloading to parsing and cleaning. All these steps can be achieved with existing functions, but the fine-tuning is specific to the set of websites. You build one (or a set of) scraping functions that take the websites as input and return a cleaned data frame ready to be analyzed.

2. You want to estimate not one but multiple models on your dataset. The models vary both in terms of data input and specification. Again, based on existing modeling functions you tailor your own, allowing you to run all these models automatically and to parse the results into one clean data frame.

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+      OPERATIONS
+      return(VALUE)
+    }
```

- We write functions to apply them later. So, we have to give them a name. Here, we name it "`my_func`".
- Also, our function (almost) always needs input, plus we want to specify how exactly the function should behave. We can use arguments for this, which are specified as arguments of the `function()` function.

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+       OPERATIONS
+       return(VALUE)
+    }
```

- Next, we specify anything we want the function to to.
- This comes in between curly brackets, `{ … }`.
- Importantly, we can recycle arguments by calling them by their name.

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+      OPERATIONS
+      return(VALUE)
+    }
```

- Finally, we specify what the function should return.
- This could be a list, data.frame, vector, sentence - or anything else really.

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

- Oh, and don't forget to close the curly brackets...

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- Our function has an intuitive name.
- Also, it takes just one thing as input, which we call `temp_F`.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- We now take up the argument `temp_F`, do something with it, and store the output in a new object, `temp_C`.
- Importantly, that object only lives within the function. When the function is run, we cannot access it from the environment.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- Finally, the output is returned.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

Now, let's try out the function:

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```r
R> my_func <- function(ARGUMENTS) {
+       OPERATIONS
+       return(VALUE)
+    }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:

```r
R> fahrenheit_to_celsius <- function(temp_F) {
+    temp_C <- (temp_F - 32) * (5/9)
+    return(temp_C)
+ }
```

Now, let's try out the function:

```r
R> fahrenheit_to_celsius(451)
```

```
## [1] 232.7778
```

[1] Yes, a function to create functions. 🤯

# Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func <- function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:

```
R> fahrenheit_to_celsius <- function(temp_F) {
+   temp_C <- (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

Now, let's try out the function:

```
R> fahrenheit_to_celsius(451)
```

```
## [1] 232.7778
```

Pretty hot, isn't it?

[1] Yes, a function to create functions. 🤯

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { … }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { … }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { … }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.
- We `else()` we specify what to do if the `if()` condition is not met.

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { … }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.
- We `else()` we specify what to do if the `if()` condition is not met.
- Make R more talkative with `message()`. Future-You will like it!

```
R> temp_convert <-
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out <- (temp - 32) * (5/9)
+   } else {
+     out <- temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

# Functional programming

R is a functional language. It encourages to use and build your own functions to solve problems. Often, this implies decomposing a large problem into small pieces, and solving each of them with independent functions.

There is much more to learn about functions and functional programming. Useful resources include:

- The chapter on functions in R for Data Science.
- The section on functional programming in Advanced R.
- The R packages book, which we will turn to later in more detail. In a way, bundling functions in a package is sometimes the next logical step.

# Iteration

## The ubiquity of iteration

- Often we have to run the same task over and over again, with minor variations. Examples:
  - Standardize values of a variable
  - Recode all numeric variables in a dataset
  - Running multiple models with varying covariate sets
- A benefit of scripting languages in data (as opposed to point-and-click solutions) is that we can easily automate the process of iteration

# Iteration

## The ubiquity of iteration

- Often we have to run the same task over and over again, with minor variations. Examples:
  - Standardize values of a variable
  - Recode all numeric variables in a dataset
  - Running multiple models with varying covariate sets
- A benefit of scripting languages in data (as opposed to point-and-click solutions) is that we can easily automate the process of iteration

## Ways to iterate

- A simple approach is to copy-and-paste code with minor modifications (→ "duplicate code", → "copy-and-paste programming"). This is lazy, error-prone, not very efficient, and violates the "Don't repeat yourself" (DRY) principle.
- In R, vectorization, that is applying a function to every element of a vector at once, already does a good share of iteration for us.
- `for()` loops are intuitive and straightforward to build, but sometimes not very efficient.
- Finally, we learned about functions. Now, we learn how to unleash their power by applying them to anything we interact with in R at scale.

# Iteration with purrr

## The tidyverse way to iterate

- For *real* functional programming in base R, we can use the `*apply()` family of functions (`lapply()`, `sapply()`, etc.). See here for an excellent summary.
- In the tidyverse, this functionality comes with the `purrr` package.
- At its core is the `map*()` family of functions.


## How `purrr` works

- The idea is always to **apply** a function to **x**, where x can be a list, vector, data.frame, or something more complex.
- The output is then returned as output of a pre-defined type (e.g., a list).
- The set of `map()`-style functions is quite comprehensive; see this cheat sheet for an overview.

# Iteration with purrr: map()

The `map*()` functions all follow a similar syntax:

$$map(.x, .f, ...)$$

We use it to apply a function `.f` to each piece in `.x`. Additional arguments to `.f` can be passed on in `...`.

# Iteration with purrr: map()

The `map*()` functions all follow a similar syntax:

$$map(.x, .f, ...)$$

We use it to apply a function `.f` to each piece in `.x`. Additional arguments to `.f` can be passed on in `...`.

For instance, if we want to identify the object class of every column of a data.frame, we can write:

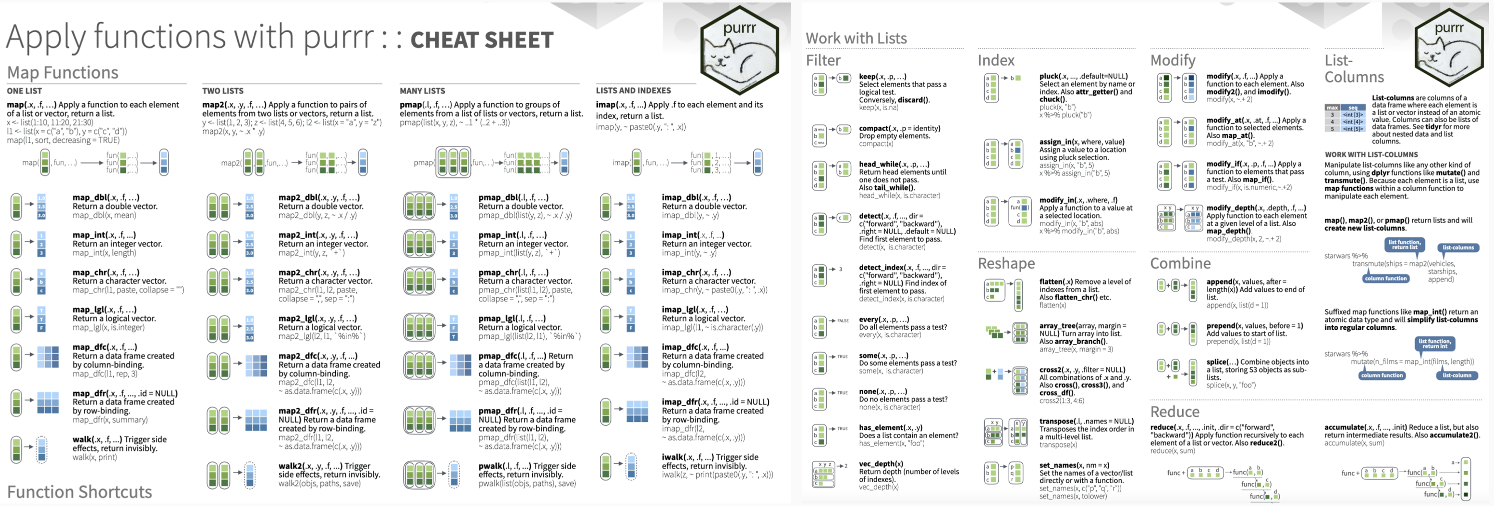
```
R> map(starwars, class)
```

```
## $name
## [1] "character"
##
## $height
## [1] "integer"
##
## $mass
## [1] "numeric"
##
## $hair_color
## [1] "character"
##
## $skin_color
```

# Iteration with purrr: map() *cont.*

By default, `map()` returns a list. But we can also use other `map*()` functions to give us an atomic vector of an indicated type (e.g., `map_int()` to return an integer vector) or a data.frame created by row- or column-binding (`map_dfr()`, `map_dfc()`).

The `purrr` function set is quite comprehensive. Be sure to check out the cheat sheet and the tutorials. You'll survive without `purrr` but you probably don't want to live with it. Together with `dplyr` it's easily the most powerful package for data wrangling in the tidyverse. If you master it, it will save you a lot of time and headaches.



# Coding style

# Coding style: the basics

## Why adhering to a particular style of coding?

- It reduces the number of arbitrary decisions you have to consciously make during coding. We make an arbitrary decision (convention) once, not always ad hoc.
- It provides consistency.
- It makes code easier to write.
- It makes code easier to read, especially in the long term (i.e. two days after you've closed a script).

# Coding style: the basics

## Why adhering to a particular style of coding?

- It reduces the number of arbitrary decisions you have to consciously make during coding. We make an arbitrary decision (convention) once, not always ad hoc.
- It provides consistency.
- It makes code easier to write.
- It makes code easier to read, especially in the long term (i.e. two days after you've closed a script).

## What are questions of style?

- Questions of style are a matter of opinion.
- We will mostly follow Hadley Wickham's opinion as expressed in the "tidyverse style guide".
- We'll consider how to
  - name,
  - comment,
  - structure, and
  - write.

# Naming things

**Surprisingly many things can go wrong with naming...**

"There are only two hard things in Computer Science: cache invalidation and naming things." - *Phil Karlton*

`Credit` karlton.org


`Credit` Mashable

# Naming files

- Code file names should be meaningful and end in `.R`.
- Avoid using special characters in file names. Stick with numbers, letters, dashes (`-`), and underscores (`_`).
- Some examples:

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

- If files should be run in a particular order, prefix them with numbers:

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

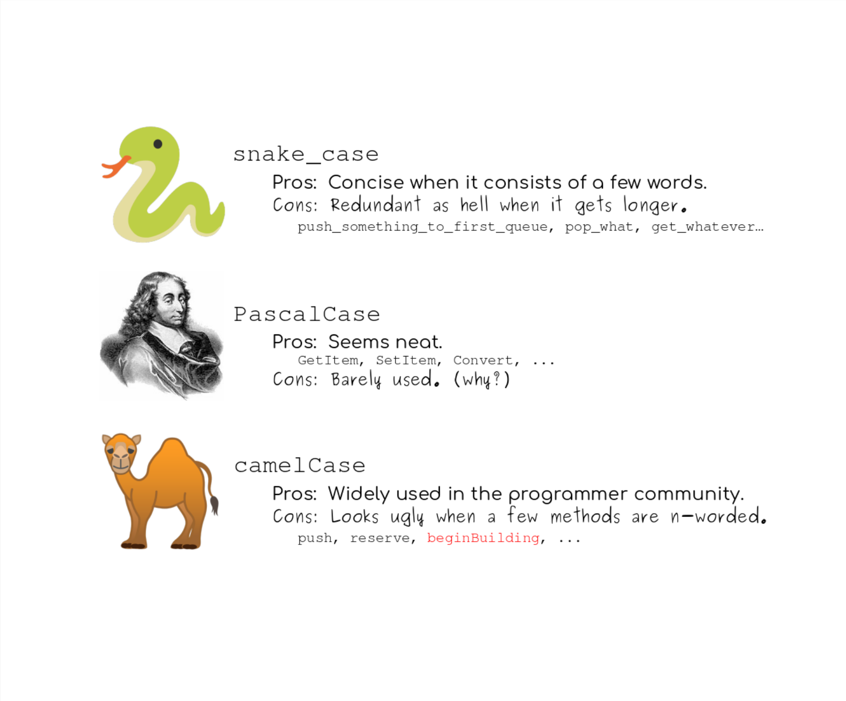
# Naming objects and variables

- There are various conventions of how to write phrases without spaces or punctuation. Some of these have been adapted in programming, such as camelCase, PascalCase, or snake_case.
- The `tidyverse` way: Object and variable names should use only lowercase letters, numbers, and underscores.
- Examples:

```
# Good
day_one # snake_case
day_1 # snake_case

# Less good
dayOne # camelCase
DayOne # PascalCase
day.one # dot.case

# Dysfunctional
day-one # kebab-case
```




`Credit` cassert24/Reddit

# Naming functions

- In addition to following the general advice for object names, strive to use verbs for function names:

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()
```

- Also, try avoiding function names that already exist, in particular those that come with a loaded package.
- This often implies a trade-off between shortness and uniqueness. In any case, you would try to avoid situations that force you disambiguate functions with the same name (as in `dplyr::select`; see "R packages").
- Check out this Wikipedia page or this Stackoverflow post for more background on naming conventions in programming!

# Commenting on things

## Why commenting at all?

- It’s often tempting to set up a project assuming that you will be the only person working on it, e.g. as homework. But that's almost never true.
- You have project partners, co-authors, principals.
- Even if not, there's someone else who you always have to keep happy: Future-you.
- Comment often to make Future-you happy about Past-you by document what Present-You is doing/thinking/planning to do.

Past-you

Present-you

Future-you

# Commenting on things *cont.*

## General advice

- Each line of a comment should begin with the comment symbol and a single space: `#`
- Use comments to record important findings and analysis decisions.
- If you need comments to explain what your code is doing, consider rewriting your code to be clearer.
- But: comments can work well as "sub-headlines".
- If you discover that you have more comments than code, consider switching to R Markdown.
- (Longer) comments generally work better if they get their own line.

```
R> # define job status
R> dat$at_work ← dat$job %in% c(2, 3)
R> dat$at_work ← dat$job %in% c(2, 3) # define job
```

## Giving structure

- Use commented lines together with dashes to break up your file into easily readable chunks.
- RStudio automatically detects these chunks and turns them into sections in the script outline.

```
R> # Input/output ---------------------
R>
R> # input
R> c("data/survey2021.csv")
R>
R> #  output
R> c("survey_2021_cleaned.RData",
+   "resp_ids.csv")
R>
R> # Load data -----------------------
R>
R> # Plot data -----------------------
```

# Other stuff

- Use **spaces** generously, but not too generously. Always put a space after a comma, never before, just like in regular English.
- Use `<-`, not `=`, for **assignment**.
- For **logical operators**, prefer `TRUE` and `FALSE` over `T` and `F`.
- To facilitate readability, **keep your lines short**. Strive to limit your code to about 80 characters per line.
- If a **function call is too long** to fit on a single line, use one line each for the function name, each argument, and the closing bracket.
- Use **pipes**. When you use them, they should always have a space before it, and should usually be followed by a new line.

**Spacing**

```
R> # Good
R> mean(x, na.rm = TRUE)
R> height <- (feet * 12) + inches
R>
R> # Bad
R> mean(x,na.rm=TRUE)
R> mean ( x, na.rm = TRUE )
R> height<-feet*12+inches
```

**Piping**

```
R> babynames %>%
+   filter(name %>% equals("Kim")) %>%
+   group_by(year, sex) %>%
+   summarize(total = sum(n)) %>%
+   qplot(year, total, color = sex, data = .,
+         geom = "line") %>%
+   add(ggtitle('People named "Kim"')) %>%
+   print
```

# Summary

# FAQ

**Q: How much time should I invest to learn the tidyverse?**

A: A week clearly is not enough. You will automatically practice more over the course of the semester. Coding is also self-learning, though. Look out for other tidyverse packages that sound interesting, and practice them!

**Q: Should I still learn base R?**

A: You are going to, automatically. All I've done is to nudge you to a certain preference. But base R is not evil. It's just a bit less accessible.

**Q: Does the tidyverse also work for Big Data**

A: Sure! However, when dealing with large datasets, you might want to consider the `data.table` package as an alternative to `dplyr`. Or just use `dtplyr`, a `data.table` backend for dplyr that allows you to write dplyr code that is automatically translated to the equivalent, but usually much faster, data.table code.

**Q: What from the tidyverse should I learn next?**

```
R> sample(tidyverse_packages(), 1)
```

# Coming up

## The first **real** assignment

Now we get serious: Assignment 2 is up on GitHub Classroom. Check it out and solve problems with the tidyverse.

## Next lecture

Relational databases and SQL. Buckle up and bring coffee, because it'll get both exciting and tedious at the same time.